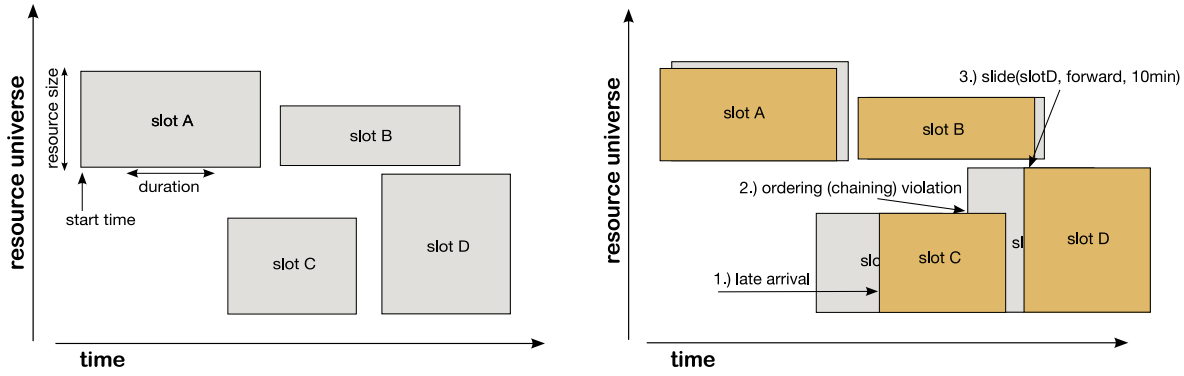
Slotted Virtual Grids

Yang-Suk Kee, Carl Kesselman, and Ken Yocum
Information Sciences Institute & UCSD Department of Computer Science

1 Introduction

This document describes an enhanced virtual grid architecture for specifying, finding, and binding resources across space and time. Version 1.0 of the virtual grid supported time-shared or dedicated resources, but most grid resources are space shared (behind batch queues) or (in the near future) available through a provisioning interface (Sisyphus [1]). Here we propose changes to accommodate resources that arrive and depart according to information in the virtual grid request; we call this the *slotted* virtual grid (SVG).

These enhancements support dynamic workflow scheduling across resources available at different points in time. Here we consider only resource access across time, not resource levels, such as the percentage CPU or memory available. For the purposes of this discussion, we assume the system has the ability of both specifying and meeting such additional constraints.



(a) Virtual grid request. (b) Virtual grid instance.

Figure 1: A virtual grid request may have multiple slots (left figure). An instance of the virtual grid evolves over time; late arriving resources (slot C) may force the application to "slide" a dependent slot forward.

The virtual grid request describes how resources arrive and depart through time. A virtual grid request may specify a *slot* for any resource set. A slot is a window of contiguous time defined by a {start, duration} tuple. Figure(a) 1 illustrates a virtual grid request with four slots. The height of the slot represents resource quantity while its length represents how long the application requires those resources.

The virtual grid execution system (vgES) provides resources that meet the constraints of kind, number, and slot. vgES delivers resources at the slot start time and removes them when the slot ends. When resources arrive at a slot, we say the slot is filled. Resource arrival is equivalent to a bound resource; it is available for job submission or interactive use through an ssh-like facility. Slots with start times in the future are said to be pending.

vgES abstracts the underlying resource management away from the virtual grid user. The underlying re-

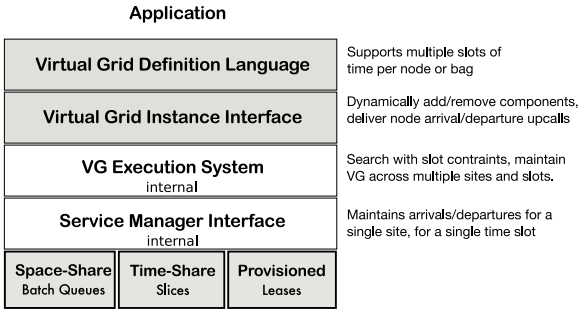


Figure 2: Four components for slotted virtual grid support.

sources for a slot may come from one or more *sites*, hosting facilities or data centers. vgES interacts with site service managers, a service that grants access to resources for some interval of time, to bind resources for a particular slot. For various reasons, resources may arrive early or late; vgES provides the necessary APIs to allow the user to adjust their resource requirements in the face of these changes.

Figure(b) 1 illustrates a virtual grid instance. Most resources arrive on schedule, and vgES delivers resource arrival and departure notifications. These are best-effort notifications; the application must manage the necessary failure and data dependencies. Note resources may arrive early to guarantee resource availability for a particular slot. However, others, like slot C, may arrive late. Such late arrivals may break the relative ordering of slot begin and end times. If slot D depends on output from slot C, the user may use the VG API to "slide" slot D to the left.

This document discusses changes (or additions to) four components of the current virtual grid architecture (Figure 2) to support slots. They are, from top to bottom:

- **vgDL:** The specification language allows users to specify one or more slots during which resources should be available.
- **VG interface:** VG API to maintain a virtual grid across multiple node arrivals, departures, and schedule disruptions (resources that do not arrive according to slot boundaries).
- **VG internals:** Internal interface and mechanisms that maintain resources across multiple sites and slots.
- **Service Manager interface:** An internal interface to the sub-system which allocates resources from a single site for a single contiguous span of time.

1.1 Slots in vgDL

To the underlying find/bind mechanism the slot start time is simply another constraint, like CPU type or cluster size. The current language supports a single {start,duration} pair or slot per virtual grid node (vgNode): single node, cluster, tightbag, or loosebag. If nested, each vgNode inherits the upper-most slot, i.e., each node in cluster inherits the cluster's slot. At any one time each vgNode in a virtual grid is assigned to one or more disjoint slots.

One may also specify multiple slots for a vgNode for resource re-use. When a user specifies multiple slots for a VG node, vgES uses the same resource constraints for each slot. Users employ this syntax to indicate and exploit data locality, relative connectivity, or to leverage a particular resource type. vgES uses this syntax as a hint that the user wants to maintain the same backing resources or exchange them for "close" ones. Thus the two following statements are equivalent:

vgNode =ClusterOf(node)[4]{node=[]} [at = 10:00 A.M. for 1 hour, at = 1:00 P.M. for 2 hours]

is equivalent to a virtual grid with two components:

c1 = ClusterOf(node)[4] {node=[]} [at = 10:00 A.M. for 1 hour] } close {
c2 = ClusterOf(node)[4] {node=[]} [at = 1:00 P.M. for 2 hours]

vgES may provide this facility by simply allocating resources from the beginning of the first slot to the end of the last. Naturally, there is a cost associated with binding resources, and, if the "off time" between slots is small, it may cost less to simply maintain the binding. Otherwise vgES will allocate the same or nearly identical resource close in space to the last slot.

2 Virtual grid instance interface

Slotted virtual grids are inherently dynamic and present an additional requirement to the underlying execution system. While one can use the existing API for creating and destroying virtual grids, `createVG`, and `terminateVG`, workload schedulers may need to see the available resources before assigning slots. Thus some uses of slotted virtual grids require separate finding and binding, already a part of the interface: `findVG` and `bindVG`. In this scenario, the application makes a slot-less vgDL request and asks the vgES to `findVG`. The application may then walk the virtual grid instance, assign slots, and initiate binding through `bindVG`.

Slot arrivals are inherently probablistic; nodes may arrive early or late for a variety of reasons: a resource manager may rescind rights to a resource, resources may crash, or resources may be taken down for maintenance. vgES provides an additional slot constraint to allow applications to bound slot arrivals to a given probability. When the user submits a virtual grid specification they may include a *slot quantile*. Slot quantiles specify the likelihood that resources arrive at or before the beginning of each slot in the vgDL specification. The slot quantile is simply an additional constraint that vgES attempts to meet across all slots in the specification. Note that a user specifies quantiles outside of vgDL and through this API.

2.1 Handling dynamism through the virtual grid API

Where resource in the previous virtual grid were all bound when instantiated, resources in a slotted virtual grid come and go. The virtual grid system provides five calls (three new) to manage an instance of a slotted virtual grid. The first two new calls inform the virtual grid user when resources arrive or depart: `transferIn` and `transferOut`. vgES issues these callbacks for all resources whether or not they arrive or depart outside of the slot boundaries.

Virtual grid clients use the other three calls to alter the virtual grid instance at runtime. The `removeNode` and `addNode` calls support clients that need to release and reallocate resources. The first call allows a client to remove pieces of a virtual grid, e.g., filled or pending slots. The second call allows a client to add additional slots.

The last call, `slide`, allows clients to adjust slots in their virtual grid, as opposed to removal and reallocation. For instance, if a computation has not completed before its slot end time, the client can attempt to *extend* the slot, keeping the bound resources for a longer period of time. [1]

Additionally, a virtual grid instance may not have all resources currently bound or all resources currently assigned to slots. The `slide` call also binds resources to slots if there is not pre-existing slot.

[1] While `slide` is the only new call in this list, the other two are unimplemented.

2.2 Example: chained slots

Here we present an example using these calls in a slotted virtual grid. Imagine that a user wants to preserve relative slot start times. They may need to do so because each successive slot depends upon the previous slot's output. We call this set of slots *chained*. Chaining preserves the duration and relative start times of all affected slots. We envision a library on top of vgES, `libchain`, that preserve chains at runtime. Like slot quantiles, vgES allows a client to specify a chain when making the original virtual grid request.

Resources that fail to arrive on time disrupt the schedule and possibly break existing chains. Once chained, `libchain` maintains the relative order of all slot start times. For example, when a resource (vgNode) arrives late, `libchain` attempts to slide all dependent slots forward such that their start times begin at the same relative offset to the end times of the violating slot. By default, the sliding approach keeps the original resources and tries to shift the start time and duration. Additionally `libchain` would export an API that allow users to add or remove chains: VGL.destroyChain() and VGL.createChain().

For example, a client may issue the following two calls:

VG vg = VGES.createVG (vgDL desc); Chain chain = VGLibChain.createChain (vg, node_list);

`libchain` receives an upcall about a late resource arrival and performs the following calls:

if (vg.slide (node, slot) == false) // fail to maintain this chain VGLibChain.destroyChain(chain); VGES.terminateVG(vg);

3 Virtual grid execution system

This section discusses internal aspects of vgES, and may be of passing interest to those outside the vgES development group. The virtual grid execution system (vgES) has three principle duties. It provides mechanisms to find matching resources, it ensures that matching resources satisfy the slot schedule, and it issues requests to underlying service managers to maintain the slot schedule.

The database schema must include the service manager type and particular service manager responsible for those resources. The finding algorithm remains roughly the same, finding multiple interchangeable candidates per node, cluster, and tight bag. If using `createVG` or `addVG`, the binding step remains integrated, and vgES ranks solution components according to their ability to fulfill their respective slots. If using `vgBind` the binding step attempts to meet the slots with the specific resources in the virtual grid instance. It returns a closest fit to the requested schedule.

Finally, vgES issues calls to the underlying service manager interface (SMI) to maintain the schedule across sites and slots. Thus it will simultaneously interact with batch queues, time-shared systems, and lease-based managers. To do so it interacts through an *internal* interface that abstracts away the management scheme exported by the particular service manager.

3.1 Internal service manager interface

vgES uses the following API calls to provide the interface to different underlying service managers. This API is internal. It may be implemented with Globus GRAM calls or calls to a lease-based provisioning service. vgES makes these calls to provide resources from a particular site for a given slot.

Note that the service manager makes no guarantees until the user calls `smAllocate`. The probe call is meant to support light-weight searches of available slots for particular node sets. Additionally, this interface includes two upcalls parallel to those provided by the virtual grid instance: `smTransferIn` and `smTransferOut`.

Service Manager Interface	
boolean smAssociate(smIP)	Initializes the local handler. May fail if service manager unavailable.
allocFD smProbe(slot,nodes,quantile)	Asks the service manager if it is possible to acquire the node set for a particular slot and quantile. Returns a handle to this request if a solution exists.
boolean smAllocate(allocFD)	Tells the service manager to allocate the node/slot pair indicated by the `allocFD`.
boolean smRemove(allocFD)	Tells the service manager to tear down the allocation.

For example, batch queue prediction would implement `smAllocate()` by submitting the appropriate number of jobs to the batch queues. Those submissions would be made to provide the specified slot quantile.

References

[1] D. Irwin, A. Yumafendi, L. Grit, D. Becker, J. Chase, and K. Yocum. Lease-based resource managment for federated systems. In *Proceedings of USENIX Technical Conference*, June 2006.